Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 00 (2015) 1–16

Procedia Engineering

www.elsevier.com/locate/procedia

The Automated Generation of Human-Comprehensible XML Test Sets

Simon Poulding, Robert Feldt

Software Engineering Research Lab,
Blekinge Institute of Technology,
371 79 Karlskrona, Sweden

Abstract

Extensible Markup Language (XML) is often used to encode complex data structures that are the inputs to software, either in the form of configuration files that the control the behaviour of the software, or the data on which the software operates. There are typically many domain-specific constraints on the hierarchy of elements in the XML, the attributes associated with each element, and the types of data that both elements and attributes contain. As a result, the automatic generation of valid XML inputs is beyond the capabilities of many test data generation techniques.

However it is not sufficient to simply generate test sets consisting of valid XML inputs: the test cases must also exhibit other properties that facilitate testing. In the absence of an automated oracle it should not be unnecessarily difficult for a test engineer to predict the correct output of a test case, and thus a desirable property of a test case is its comprehensibility by a human.

In this paper we demonstrate the use of the GödelTest framework in generating XML test inputs, and show the generation strategy can be optimised using Nested Monte-Carlo Search to produce test cases that are more comprehensible by the test engineer. Moreover, when the validity of the inputs is defined using an XML Schema definition, we show that it is possible to automate much of the generation process and thereby realise significant savings of time and effort.

Keywords: Search-Based Software Testing, Nested Monte-Carlo Search, XML

1. Introduction

The manual generation of test inputs is often time-consuming and labour-intensive, and so any degree of automation in the generation processes can free up resources that can be invested elsewhere in order to improve the quality of the developed software. When the software-under-test takes inputs that are complex data structures in the form of—for example—records, lists, trees, or graphs composed from more primitive data types, the degree of automation that is possible may be limited. These data structures are often unbounded in terms of size and typically have domain-specific constraints on the values that they may contain; both these characteristics present a challenge for many automated test data generation techniques.

In our previous work we introduced the GödelTest framework for generating test inputs [1]. GödelTest enables the test engineer to specify the construction of correctly-formed test inputs using a non-deterministic generating program, and so the framework is particularly suited to the generation of domain-specfic data structures.

However, it may not sufficient to generate test inputs that simply have a valid structure; the test engineer will often require the inputs to have other properties, the nature of which will depend on the type of testing being performed and the quality objectives that are required. When evaluating reliability, for example, the property required of the test inputs might be that they have a profile similar to that which will be experienced by the software in operation. When testing scalability, the required property might be that the test inputs have a particular size. When looking for faults in the software, the required property might be that a small set of the test inputs cover as much of the software's code as possible.

A key feature of the GödelTest framework is that it tracks and controls the execution paths taken through the generating program, and can optimise these paths so that the generating program emits test inputs that have the properties desired by the test engineer. Our previous work demonstrated how this optimisation of the test input properties may be achieved in this way by applying Differential Evolution [1] and Nested Monte-Carlo Search [2] to GödelTest.

In this previous work, the example of a complex data structure we considered was unlabelled general trees. This data structure presented a sufficient challenge to demonstrate the capabilities of the GödelTest framework, but was rather abstract in nature. In this paper we demonstrate the use of GödelTest in generating a more concrete, real-world data structure: Extensible Markup Language, more commonly known as XML.

XML is a format used by many software systems to communicate structured data. An example is Extensible Business Reporting Language (XBRL), an XML-based standard for the exchange of business information [3]. A particular advantage is that, as well as being sufficiently structured to be parsed by software, XML can also be read and edited by humans and so it is often used to specify the configuration of software systems as well as data on which the software operates. Masmano et al., for example, describe a hypervisor for safety critical systems that is configured using XML [4].

XML test inputs must not only conform to the general constraints on structure (e.g. that each opening tag must have a corresponding closing tag), but the hierarchy and type of elements, attributes, and text must also satisfy domain-specific constraints. Generating valid XML for the particular software-under-test can therefore be a difficult and expensive process that would benefit from automation.

The first contribution of this paper is an automated process that addresses the challenge of generating domain-specific XML. Not only do we show that GödelTest can automatically generate well-formed XML, but that we are also able to automate the process of creating the GödelTest generating program from the XML Schema definition, a standard specificiation language for domain-specific XML. This avoids the need for the test engineer to develop the generating program manually.

A potential limitation of any technique for automatically generating test data is that the inputs may not be easy for a human to comprehend. An example of a characteristic that limits comprehensibility is the size of test inputs that are not bounded in size; Fraser and Arcuri, for example, highlight and manage the issue of such 'bloat' in test inputs generated by a search-based technique [5].

The lack of comprehensibility can be a particular problem when the oracle—the resource used to predict the correct output of a test case—is a human rather than an automated resource such as an executable model. If the test inputs are hard for the human to comprehend, not only will the process of predicting the correct test outputs be costly, but the human is more likely to make mistakes. Even when an automated oracle is available, test inputs that are difficult to comprehend may hinder the developer in resolving any faults detected by the test case.

The second contribution of this paper is demonstrating how the XML test inputs generated by the GödelTest framework can be optimised in order to improve comprehensibility using Nested Monte-Carlo Search.

To illustrate both contributions, we perform an empirical study using MathML, an XML format for describing the presentation of mathematical expressions. We show that the generated test cases achieve similar code coverage to a manually-derived test set, even though the cost of generating the test cases is much lower when using the highly-automated GödelTest process. Although fault finding was not an explicit objective of the study, we additionally report on a number of apparent bugs that were found during this process.

The paper is structured as follows. Section 2 describes the GödelTest framework, introduces the terminology used to describe XML, and outlines the XML Schema definition language. In Section 3, we describe

```
function stringGen
    word ← mult(asciiCharGen)
    return word
end function

function asciiCharGen
    code ← choose(Int,32,126)
    char ← character with ASCII value code
    return char
end function
```

Fig. 1. GödelTest generating program that emits a string of ASCII characters

how a GödelTest generating program for XML may be automatically constructed from an XML Schema definition. Section 4 describes the empirical study using MathML. Section 5 places the contributions of this paper in the context of related work. We conclude the paper and outline future work in Section 6.

2. Background

2.1. The GödelTest Framework

We have previously introduced GödelTest, a flexible framework for generating complex test inputs [1]. The framework consists of three components: custom programs for generating test data; associated choice models; and mechanism for optimising a choice model so that the generated test data has desirable properties.

2.1.1. Generating Program

A generating program consists one or more functions. A simple example—a generating program for ASCII strings—is shown in Figure 1. (The GödelTest framework used for the research described in this paper uses generating programs written in the Julia language [6]. However, the approach is not dependent on the features of Julia, and so for clarity, the generating programs are shown here as pseudocode.)

The asciiCharGen function emits an ASCII character. In this function, **choose**(Int) is a GödelTest construct that returns an integer from the range specified by the second and third parameters. The top-level stringGen function uses the GödelTest **mult** construct to call the asciiCharGen function zero or more times. The stringGen function therefore emits a ASCII string of length zero or more.

The use of programs to generate test input is a more flexible approach than the alternative of a grammar. Generating programs can have store and act on internal state and so enable the construction of structures that cannot be represented by a grammar.

Within a generating program there will be a number of *choice points* that represent a choice of execution path or data value. In the example above, the GödelTest constructs **mult** and **choose** are both choice points: **mult** makes a choice of execution path, i.e. how many times to call the asciiCharGen function; **choose** makes a choice as to the value that will be stored in the *code* variable. It is the existence of these non-deterministic choice points that enables a single generating program to emit a range of different inputs.

2.1.2. Choice Model

The second component of the GödelTest framework is a choice model. When a choice point is encountered during the execution of the generating program, the choice model is used to determine which one of the available choices to make. Conceptually, the choice model provides a number, called a Gödel number, that identifies the choice to make. The set of Gödel numbers provided during a single execution of generating program is a Gödel sequence. Since the Gödel sequence determines the execution path and data states of the generating program, it also determines the test input emitted by the program. This relationship is illustrated

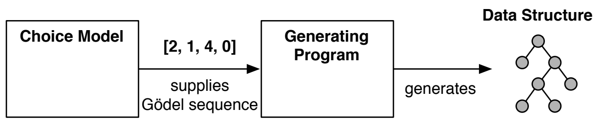


Fig. 2. The GödelTest framework.

by Figure 2. The problem of finding a test input with the desired properties is therefore abstracted to the problem of deriving an appropriate Gödel sequence.

The choice model may provide Gödel sequences in a deterministic manner to enable exhaustive testing, or sample Gödel sequences stochastically to permit probabilistic forms of testing. It is the latter approach that is considered in this paper and a 'sampler' choice model is used for this purpose. In this model, each choice point is associated with a probability distribution. For example, the **mult** choice point is associated with a geometric distribution in the model: this distribution samples a non-negative integer, and small integers are more likely to be sampled than large integers. The **choose**(Int) choice point is associated with a discrete uniform distribution that samples an integer in range specified by arguments to the construct.

2.1.3. Optimisation for Desirable Properties

If a desired property of the test inputs can be quantified as a metric, then GödelTest can use this metric to optimise the generation process to favour test inputs with the property. Since GödelTest abstracts the choice model from the generating program, the optimisation is applied to the choice model.

In our previous work, we have described two complementary mechanisms for optimising a sampler choice model in this way.

The first mechanism applies metaheuristic search to the parameters of the choice model [1]. Most of the probability distributions in a sampler choice model have one or more parameters that control the likelihood of sampling particular Gödel numbers. For example, the geometric distribution (associated with a **mult** choice point) has single parameter that takes values between 0 and 1; when the parameter is closer to 0, larger Gödel numbers are more likely. Changing these model parameters thus affects the probability distribution over the domain of test inputs, and so the model parameters can be optimised to favour inputs that have the desired property. This optimisation occurs once, *prior* to generating a set of test inputs.

In contrast, the second form of optimisation occurs *during* the generation of each test input. It applies Nested Monte-Carlo Search (NMCS), an algorithm used to solve single-player games, to the selection of each Gödel number. We provide here an overview of how NMCS is used for this purpose, and refer the reader to [2] for further details.

NMCS is normally applied to the 'game tree'—the tree of possible future moves—in a single-player game such as solitaire. To assesses which is the best move to make next, NMCS considers each of the these moves in turn, and starting from each move, plays the game to completion in a 'simulation'. During the simulation the moves are played according to a simple policy such as choosing moves at random, or alternatively using NMCS itself (hence the nested nature of the approach). The terminal game state is then evaluated according to a chosen metric. Whichever of the top-level moves from the current game state led to the best value of this metric at a terminal game state in simulation is then taken. Figure 3 illustrates this technique.

While it would appear that the 'signal' returned from a single simulation from each of the possible moves from the current game state would be hard to detect among the 'noise' arising from the random nature of the simulations, NMCS and other types of Monte-Carlo Tree Search are surprisingly effective at winning games, i.e. optimising the metric that assesses the terminal game state [7].

To apply this search technique to GödelTest, we equate game states to choice points, and moves to the possible choices, identified by Gödel numbers, at a choice point. When a Gödel number is requested from the choice model, a small set of Gödel number are sampled from the probability distribution associated with the choice point. Each of the Gödel numbers are assessed in turn using a 'simulation' that first applies that Gödel number to the choice point, and then runs the generating program to completion with the Gödel numbers for the remaining choice points encountered in the program sampled at random from the choice

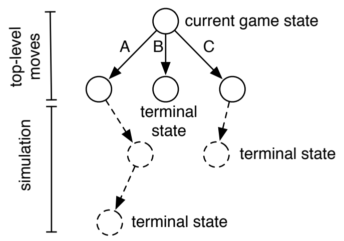


Fig. 3. An example of Nested Monte-Carlo Search for a single-player game.

```
<book inPrint="true">
 <title>The C Programming Language</title>
 <author>
  <firstname>Brian</firstname>
  <lastname>Kernighan</lastname>
 </author>
 <author>
  <firstname>Dennis</firstname>
  <lastname>Ritchie</lastname>
 </author>
</book>
```

Fig. 4. An example of XML containing information about a book.

model. The Gödel number that results in the 'best' candidate input by simulation, as determined by the property metric, is the one returned by the choice model for the original choice point, and the execution of the generating program continues on to the next choice point. The process can be nested so that during the simulation process, NMCS itself is used rather than random sampling; this can lead to more effective optimisation at the expense of a larger number of simulations and so more computational effort required to generate a single test input.

In [2], we found that NMCS was an efficient mechanism for generating tree structures with specific properties. In particular, it can be faster than simply generating trees at random and rejecting those that did not have the desired properties. Since XML has a tree-like structure, we apply NMCS as the optimisation mechanism in this paper. We use GT-NMCS to denote the combination of the GödelTest framework and NMCS.

2.2. Extensible Markup Language

2.2.1. Structure and Terminology

A simple example of Extensible Markup Language (XML), containing the information about a book, is shown in Figure 4. A pair of opening and closing tags delimits an *element*: `<title>` ... `</title>` is an instance of a `title` element. Elements can contain other elements: for example, the `author` element contains `firstname` and `lastname` elements. Elements can also contain text between the tags; each contiguous piece of text, i.e. uninterrupted by elements, is a *text node*. Thus `The C Programming Language` is a text node that is a child of the `title` element. Elements may have *attributes* associated with them using a `name="value"` notation in the opening tag.

2.2.2. XML Schema Definition

XML Schema Definition (XSD) is one of a number of methods of specifying domain-specific XML formats, and is a recommendation (i.e. standard) of the World Wide Web Consortium (W3C) [8]. XSD is itself an XML format which specifies the valid elements and attributes that the domain-specific XML can

```
<xs:element name="book">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="title" type="xs:string"/>
   <xs:element ref="author" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="inPrint" type="xs:boolean"/>
 </xs:complexType>
</xs:element>
```

Fig. 5. A fragment of XSD specifying the book element.

contain, the data types permitted in text nodes and attribute values, and how the elements may be arranged as a hierarchy.

A fragment of the XSD for the book example discussed above is shown in Figure 5. This fragment specifies the constraints on the `book` element: it must contain exactly one `title` element followed by one or more (indicated by `maxOccurs="unbounded"`) `author` elements. The `book` element can take one attribute, `inPrint`, and by default, the use of this attribute is optional. The `title` element can contain a single text node that is a string of Unicode characters: this indicated by the 'built-in' data type `xs:string`. The `inPrint` attribute has the built-in data type `xs:boolean` which consists of the values `true` and `false`. The `author` element is defined by a reference to its type that is defined elsewhere in the XSD (and not shown in this fragment).

3. Automating the Construction of Generating Programs for XML

GödelTest generating programs for domain-specific XML may need to incorporate a large number of constraints, and thus the substantial effort may be expended in constructing the program. In this section we describe a 'connector', XSD-to-GT, that can automate much of the construction of the generating program. The process is illustrated in Figure 6: XSD-to-GT uses the XSD to automatically construct a generating program that incorporates the domain-specific constraints. The GT-NMCS algorithm is then used to generate XML test inputs with desirable properties using this generating program.

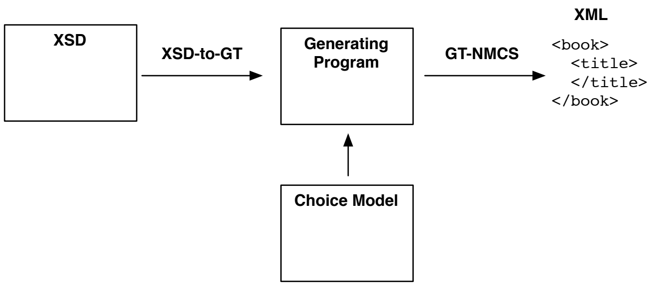


Fig. 6. The generation process using the XSD-to-GT connector.

The process is portrayed as linear in the figure but in practice it is likely to be iterative. Any faults identified by executing the generated test cases could be the result of errors or omissions in the XSD specification rather than faults in the software itself. After correcting any faults in the XSD, the XSD-to-GT connector permits the generating program to reconstructed automatically.

3.1. The XSD-to-GT Connector

The XSD-to-GT connector used for the research described here is implemented as a script of approximately 1,500 lines of Julia code and uses the Julia LightXML package [9] to parse the XSD. The current

```
function bookElementGen
    elmt ← new book element
    elmt ← elmt + titleElementGen
    elmt ← elmt + plus(authorElementGen)
    if choose(Bool) then
        elmt ← elmt + inPrintAttributeGen
    end if
    return elmt
end function

function titleElementGen
    elmt ← new title element
    elmt ← elmt + stringGen
    return elmt
end function

function inprintAttributeGen
    attr ← new inPrint attribute
    attr ← attr + booleanGen
    return attr
end function
```

Fig. 7. The GödelTest functions automatically constructed by the XSD-to-GT connector for the XML Schema fragment of Figure 5.

implementation of the connector handles the most common XSD constraints, most built-in types, and custom types defined by enumerations and XSD patterns (regular expressions).

Figure 7 shows the functions that XSD-to-GT would construct for the XSD fragment of Figure 5. (As above, we omit implementation details and show the generating program as pseudocode for clarity.) Many XSD constraints map naturally to GödelTest constructs. For instance, `maxOccurs="unbounded"` in the XSD fragment specifies that the `book` element may have one or more child `author` elements: this is realised in the generating program through the use of the **plus** construct which calls the authorElementGen function one or more times. The `inPrint` attribute is optional for the book element, and so a **choose** construct is used to return a Boolean value that determines whether the inPrintAttributeGen function is called.

Figure 7 does not show the authorElementGen function itself. The XSD specifies the `author` element by reference to a type, and thus this function will be created from the type definition elsewhere in the XSD. Functions that emit XSD built-in types, such as `xs:string` and `xs:boolean`, are created as required by the connector.

The XSD-to-GT connector outputs the generating program as a Julia source code file. This permits the manual refinement of the program as described below, as well as facilitating storage and version control in the same manner as other test-related files.

3.2. Manual Refinement

It may be necessary for the test engineer to edit the source of the generating program to refine the generation logic. We give two examples of situations in which this may be required.

The first is when the XSD specification cannot completely specify the structure of valid data. For example, if an attribute in the XML should contain the URL of a webpage, the XSD can specify the pattern that an URL must conform to (as a regular expression), but cannot specify how to generate URLs for *existent* webpages. The test engineer may therefore need to amend the generating program to output URLs of some webpages that are known to exist.

The second example is a semantic relationship between parts of the XML that are generated independently. Consider an XML format that is used to exchange an invoice between supplier and customer. Each

```
<math mathcolor="green">
 <mi>m</mi>
 <mo>=</mo>
 <mfrac>
  <mi>E</mi>
  <msup><mi>c</mi><mn>2</mn></msup>
 </mfrac>
</math>
```

Fig. 8. Presentation MathML for displaying the equation $m = \frac{E}{c^2}$ in green.

invoice line may be represented by an XML element which has the line total as an attribute. Elsewhere in the XML an invoice total may be specified that, for a valid invoice, should equal the sum of the invoice line totals. Such a relationship cannot be enforced through XSD, and is unlikely to occur by chance when the line and invoice totals are generated independently.

In order to ensure that at least some of the test cases represent valid invoices, the test engineer can enhance the generating program to enforce the relationship between line and invoice totals. The use of generating programs—rather than a grammar—in the GödelTest framwork facilitates this: the line totals may be passed between functions and additional processing to calculate the sum of the line totals can be added to the code of the relevant function.

4. Empirical Study

In this section we describe an empirical study that assesses:

RQ1 To what extent can the automated process proposed in Section 3 be realised in practice?

RQ2 Can GT-NMCS be used to improve the comprehensibility of the generated XML inputs?

RQ3 How effective are the XML test sets generated by GödelTest in terms of code coverage, even though coverage is not a property that is explicitly optimised for?

4.1. Software-under-Test and XML Format

For the empirical study, the software-under-test is JEuclid (version 3.1.9) [10], a Java library that renders mathematical expressions expressed in the Mathematical Markup Language (MathML) 2.0, a domain-specific XML format [11]. There are two forms of MathML: one focuses on communicating the semantics of the expression (content MathML) and the other on the display of the expression (presentation MathML). It is the latter that we consider in this study.

An example of presentation MathML is shown in Figure 8. The elements `mi`, `mo`, and `mn` identify their child text nodes as variables (indicators), operators, or numbers, respectively, so that they may be rendered appropriately. A large set of additional elements, such as `mfrac` (fraction) and `msup` (superscript), control other aspects of the layout. Most elements have many optional attributes that customise the presentation further by, for example, controlling the colour, font, and alignment.

We selected MathML for the empirical study for the following reasons.

- It has many different element and attributes types, and encapsulates information in number of different ways: in the hierarchy and ordering of elements, in the attributes of each element, and in text nodes associated with the `mi`, `mo`, and `mn` elements. This enables us to argue some generality in results of this study in terms of type of XML to which the process can be applied, and the scalability of the process itelf.

- MathML has a significant real-world use: to display mathematical expressions in web pages. It is a recommendation (standard) of the World Wide Web Consortium (W3C) for this purpose [11], and modern browsers support MathML, either natively or using JavaScript libraries.
- W3C provide an XSD specification for MathML 2.0 which utilises many features of the XML Schema language and so this XSD will exercise much of the functionality of the XSD-to-GT connector.
- W3C provide a manually-constructed test set for MathML 2.0 against which the test sets generated by GödelTest can be compared.

4.2. Comprehensibility Metric

In our previous work that generated random tree structures [1], it was found that in the absence of any optimisation of the generation process, many of the generated trees were small (i.e. had few nodes), while others were very large. A similar distribution of tree sizes was also seen when using Boltzmann samplers [12], a generation technique based on combinatorics and with a strong analytical foundation, and so we speculate that this distribution of sizes is a consequence of the tree data structure rather than of the generation technique. XML inputs have a tree-like structure, and if a similar size distribution were to arise in the XML inputs generated by GödelTest, this would have two consequences for the efficiency of the testing process.

Firstly, small and simple XML inputs would be unlikely to exercise much of the code in the software-under-test, and therefore be unlikely to find many faults. The costs incurred in executing and checking such a test case may be wasted.

Secondly, large and complex XML inputs may be difficult for test engineer to comprehend. The effort involved in predicting the correct output of the software for this input, and checking that the observed output matches this prediction may be substantial. As an extreme example, it might be possible to achieve high code coverage of the software using a single test case with a very large XML input. It would be difficult, however, for a test engineer to fully comprehend such an input and to correctly predict the output. Moreover, should this single test case detect a fault in the software, it may be of little assistance to developer in determining the location of the fault.

We argue that there is an optimal range of complexity for a single XML input: it must be sufficiently complex to exercise the software so that faults may be found, but not too complex for the test engineer to comprehend.

In the context of MathML, we propose the following quantities affect comprehensibility of a test input:

- the number of elements, n_{elmt};
- the number of attributes, n_{attr}; and,
- the number of text nodes[1], n_{text}.

Our argument is that each additional element, attribute and text node adds complexity to the presentation of the MathML, and thus to the effort required by the test engineer to comprehend it and predict the correct output. For example, a new `mo` element adds a further operator to the expression; a new `mathcolor` attribute may change the colour of the part of the expression, while a new text node in, for example, an `mn` element adds further text to be displayed.

The three values may used to guide GT-NMCS as the following single fitness function:

$$f = \sqrt{\left(\log \frac{n_{\text{elmt}}}{t_{\text{elmt}}}\right)^2 + \left(\log \frac{n_{\text{attr}}}{t_{\text{attr}}}\right)^2 + \left(\log \frac{n_{\text{text}}}{t_{\text{text}}}\right)^2} \tag{1}$$

[1]The Julia LightXML package used in this paper to output the XML test inputs adds whitespace between elements to make it more human readable. In the empirical study of Section 4 we therefore count only text nodes that do not consist only of whitespace.

where t denotes the target value of corresponding metric. The target values are likely be domain-specific, and so can be set by test engineer depending on the nature of the XML format.

An optimal XML input for which the observed metrics equal the target values will have a fitness of zero and the fitness takes increasingly larger positive values the further away the observed metrics are from the targets. By taking the ratio of the observed and target values, the contribution of each observed metric to the overall fitness is normalised. By taking the logarithm of the ratio and then squaring it, the same fitness is calculated when the observed metrics is, for example, half the target value as when the it is twice the target value. The use of the logarithm also has the useful property of severely penalising MathML inputs having, for example, no text nodes, by returning an infinite fitness: such an input would typically cause no math expression to be rendered by the software, and thus is unlikely to be an effective test case.

The argument that we make for the appropriateness of this comprehensibility metric in the context of MathML—that each additional element, attribute, and text node typically increases the complexity of the rendered expression and therefore also increases the effort the human engineer must expend in checking its correctness—is likely to apply to XML used in other domains. We therefore speculate that the metric may have application beyond this particular case study.

4.3. Preparation

Prior to experimentation, the W3C XSD for presentation MathML was modified as follows.

- The W3C test sets include most, but not all, of the elements and attributes of presentation MathML. To enable a fair comparison between the test sets generated by GödelTest and the W3C test sets, any element or attribute not present in at least one of the W3C test cases was removed from the XSD. This is a conservative decision that is likely to favour the W3C test set in any comparison.
- A few attributes in the XSD do not have a fully-specified type. For example the attribute `mathcolor` should specify a valid colour name, or an RGB hex value, but the XSD permits any string. (A comment in the XSD notes this problem.) The types of these attributes were updated so that meaningful values could be generated.
- A bug was identified in the XSD type patterns (regular expressions) for attributes that specify a length and associated unit. The definition was amended to fix this bug.
- A bug was identified that incorrectly specified an attribute group of `mi.attlist` for the `mn` element: this was corrected to use the `mn.attlist` group.
- JEuclid was found to raise a Java exception when the a `mathalign` element was used in some parts of the element hierarchy. To prevent this apparent bug in JEuclid from affecting coverage measurements, the part of XML Schema definition giving rise to this problem was removed.

These changes were made only to ensure a fair comparison agains the manually-constructed W3C test set; most of them would not normally be made in practice.

4.4. Process Automation (RQ1): Method and Results

The XSD-to-GT connector was successful in constructing a GödelTest generating program from the MathML XSD. As an indication of the complexity of the XSD specification for MathML, the generating program consisted of approximately 3,000 lines of Julia code and 569 functions. This would have been very time-consuming and error-prone for a test engineer to construct manually.

The connector took approximately 5 seconds to construct this generating program on a computer with a 1.3 GHz processor and 8 GB of memory.

No manual refinement (Section 3.2) of the generating program was necessary in this case.

4.5. Comprehensibility (RQ2): Method and Results

First, 30 test sets, each consisting of 208 MathML test cases, were generated using the GT-NMCS algorithm and the comprehensibility fitness metric of equation (1). The test set size of 208 is the same as that of the W3C test set for presentation MathML, and was chosen to enable these test sets to be re-used in the experiment below that assesses RQ3.

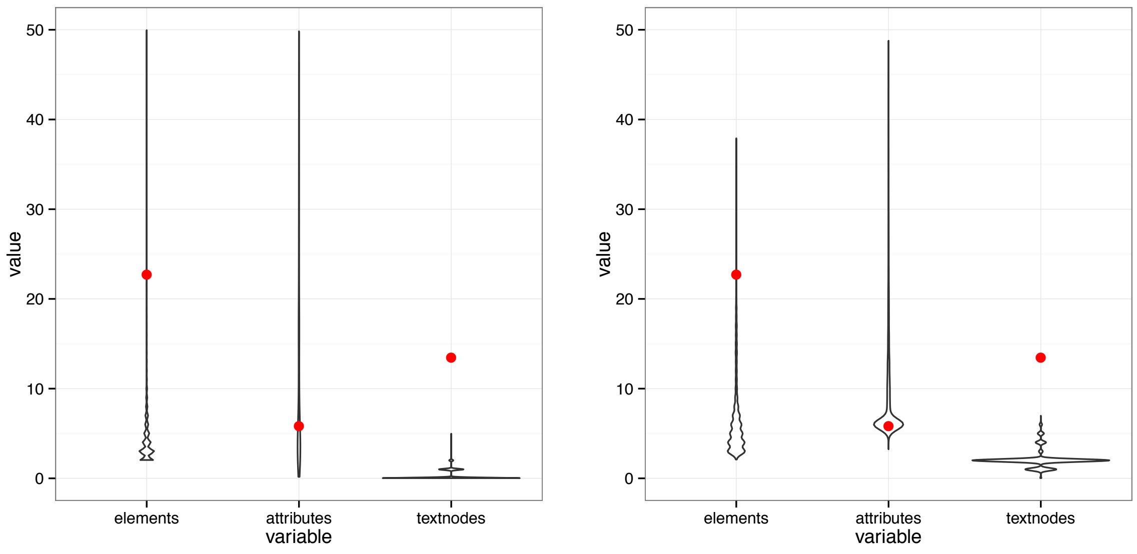
For this study, the target values for the number of elements, attributes and text nodes in the metric were set objectively by measuring the mean values of these properties in test cases of the W3C test set. This gave target values of $t_{\text{elmt}} = 22.72$, $t_{\text{attr}} = 5.80$, and $t_{\text{text}} = 13.43$. In practice, the target values would be set by the test engineer depending on the degree of comprehensibility required.

The GT-NMCS algorithm takes two parameters. The nesting level was set to 1, and the number of Gödel numbers assessed at each choice point during simulation was set to 4. These parameter settings had been found, in previous work, to demonstate an acceptable trade-off between the performance of the algorithm and the optimality of the test inputs in terms of the desirable properties [2]. With these parameters, a test set of size 208 could be generated using GT-NMCS in approximately 240 seconds.

Next, 30 test sets, again each of 208 test cases, were generated by simple random sampling from the GödelTest choice model (we denote this process GT-Random). These test cases give an indication of the test inputs that would occur were the generation process not optimised for comprehensibility.

For all test cases from both GT-NMCS and GT-Random, the number of elements, attributes, and text nodes was counted. These results are shown as violin plots in Figure 9. The width of the plot indicates the frequency at different counts. The red points indicates the corresponding target values (derived from the W3C test set).

To enable an easier visual comparison, the y-axes of the two violin plots range from 0 to 50, but the upper limit truncates the plots of element and attribute counts, particularly of GT-Random. For GT-Random, the largest element count was 86, and 8 of the 6240 (30 sets of 208 cases) randomly sampled inputs had more than 50 elements; the largest attribute count was 292, and 205 of the inputs had more than 50 attributes. For GT-NMCS, the largest attribute count was 62, and 3 of the inputs had more than 50 attributes.



(a) GT-Random (b) GT-NMCS

Fig. 9. The distribution of element, attribute, and text node count in test cases generated by (a) GT-Random and (b) GT-NMCS.

4.6. Code Coverage (RQ3): Method and Results

JEuclid can output rendered MathML in a variety of image formats, but for simplicity, the test sets are assessed using the coverage of the core code (i.e. that contained in `jeuclid-core-3.1.9.jar`) when rendering MathML to a single image format: SVG. The Java Code Coverage (JaCoCo) library of the EclEmma coverage tool [13] is used to measure coverage in terms of the proportion of the bytecode instructions exercised by the test set.

The manually-derived W3C test set covered 61.6% of the core code.

The coverage was measured of each of the 30 test sets generated using GT-NMCS for RQ2 in the preceding experiment. The coverage induced by these test sets varied between 58.0% and 60.2%, with a median of 59.3%. The difference from the coverage of W3C is statistically significant: a one-sample Wilcoxon test calculates a p-value of less than 10^{-5}.

4.7. Discussion

4.7.1. RQ1: Process Automation

For presentation MathML, the automated process proposed in Section 3 was realised in practice. The XSD-to-GT connector was able to automatically construct a generating program and associated choice model; and from these, GT-NMCS could automatically generate XML test inputs.

The time taken to construct a GödelTest generating program from the XSD was approximately 5 seconds, and the time taken to generate one test set of 208 inputs from the generating program was approximately 240 seconds. Thus, in an ideal process, an optimised test set could be produced from the XSD in under 5 minutes. (This assumes, of course, that the XML Schema definition already exists, as was the case here.) In practice, the process was iterative and took longer since both the automatic construction of the generating program and the generation of test inputs revealed errors in the XSD specification.

4.7.2. RQ2: Human-Comprehensible XML Inputs

The results illustrated in Figure 9a for GT-Random show that, in the absence of a metric guiding the generation process, many small test inputs are generated. Many of the test inputs have one element, no attributes or no text nodes and will therefore exercise little of the code. Conversely some test inputs have 50 or more attributes—compared to an average of 5.8 in the manually-derived W3C test set–and such MathML will be difficult for a human to comprehend and predict the correct rendered output.

The results illustrated in Figure 9b show that GT-NMCS, using the fitness metric of equation (1), can generate MathML with properties closer to the target values. Fewer of the inputs have only one element, no attributes or no text nodes. In particular, many more of the inputs have at least one text node compared to GT-Random. No inputs have more than 38 elements, nor more than 62 attributes; indeed many of the inputs have a number of attributes close to the target value indicted by the red point, and are therefore more comprehensible.

Nevertheless, most element counts are below the target value and many of the attribute counts are above it. The GT-NMCS algorithm appears to have difficulty in increasing the element count while simultaneously reducing the attribute count. In MathML, most elements have many optional attributes, and the default choice model includes each of these attributes in the XML with a probability of 0.5. We speculate that this leads to a highly variable number of attributes in candidate inputs produced during each simulation performed by the NMCS algorithm, and this variability prevents the algorithm making the correct choice as to whether or not to include each individual attribute. This causes too many attributes to be included in the test input, or—to compensate—the algorithm includes fewer elements and thus avoids the potential to add more attributes.

In MathML, text nodes are only output for the `mi`, `mo`, `mn`. These nodes are relatively deep in the hierarchy of XML elements, and we speculate this is the reason why the inputs generated GT-NMCS all have a text node count below the target.

The results demonstrate that proposed metric is useful in permitting GT-NMCS to produce XML inputs that have a comprehensibility closer to the target values, but the optimisation technique is not as effective as we would like.

Table 1. Possible bugs found in the JEuclid software and/or the MathML XSD specification.

Location	Effect	Apparent Cause
JEuclid (or XSD)	IllegalArgumentException	`fontfamily` attribute specifies an invalid font
JEuclid (or XSD)	IndexOutOfBoundsException	a particular use of the `malignmark` element
JEuclid (or XSD)	Unsupported notation message	some values of the `notation` attribute
JEuclid (or XSD)	Invalid value warning	some values of the `rowalign` attribute
JEuclid (or XSD)	Number parsing warning	length attributes with values that start with a decimal point
JEuclid (or XSD)	Number parsing warning	`maxsize` has a negative value
XSD	Invalid length attributes	incorrect patterns for `length-with-[optional-]unit` types
JEuclid (when outputting PNG)	OutOfMemoryError	`mathsize` attribute specifies a large size
JEuclid (when outputting PNG)	OutOfMemoryError	`scriptlevel` attribute is negative

4.7.3. RQ3: Code Coverage

The test sets generated using GT-NMCS cover the core code of JEuclid to almost the same extent as the manually-derived W3C test set. This is perhaps an unexpected result given that coverage was not a property that was optimised for during the test data generation process. Moreover, if there is a trade-off between comprehensibility and coverage—for example, because larger, and so less comprehensible inputs, generally exercise more of the code—optimising for comprehensibility could have led to a reduction in coverage.

We interpret this result positively as showing that generating comprehensible XML using GT-NMCS *can* be competitive with manually-derived deterministic tests in terms of coverage, but we obviously require further experimentation on additional SUTs in order to draw a general conclusion. We speculate, for example, that the JEuclid core code may be amenable to testing by randomly-generated data of the type emitted by GT-NMCS since the range of different XML elements and attributes in the test set is likely to be a more important factor in determining code coverage than the specific values of attributes and text nodes (expect for the operators discussed below).

A possible explanation for the slightly lower coverage achieved by GT-NMCS is that JEuclid, like most renderers of MathML, may have special handling that formats some common operators, such as $\sum$ and $\int$, in custom manner when these symbols appear in an `mo` element. However the XSD specifies the text node in the `mo` element as having a string type, and so these special symbols are unlikely to occur by chance in the XML generated by GödelTest. In practice, the test engineer could incorporate this knowledge about the program by amending the XSD or manually refining the GödelTest generating program.

4.8. Bugs Found

Finding faults was not an explicit objective of this empirical study. However, a number of possible bugs were identified when preparing and running the study. In many cases it is unclear whether the fault lies in the JEuclid software or the XSD specification. The bugs are listed in Table 1.

5. Related Work

5.1. Generating Structured Test Inputs

Automated techniques for generating structured test inputs typically use either non-deterministic generating programs or grammars. The algorithms of Beyene and Andrews [14], and of Poulding et al. [15] are examples of the grammar-based approach that are particularly notable: both use metaheuristic optimisation to generate random test cases with desirable properties.

However grammar-based approaches are less flexible than non-deterministic generating programs of the type used by GödelTest. UDITA [16] is a Java-based framework that generates test sets by exhaustively enumerating all points of non-determinism in the generating program up to a chosen bound. QuickCheck [17] is a framework for testing programs in Haskell and other languages by generating data at random. A key difference between QuickCheck and GödelTest is that manual optimisation must be performed for QuickCheck in order to generate test inputs with desirable properties, while GödelTest can perform this optimization automatically.

5.2. Generating XML

Bertolino et al. proposed an XML-based partition testing (XPT) method that derives XML instances from schemas [18] and in later papers used XPT to test web services [19] and access control policies [20]. Since the method systematically exhausts the alternatives described by the schema it suffers from an explosion in the number of instances generated. The authors extended their tool with element weighting and strategies to select only a subset of the schema in a specific derivation. However, it is unclear how these weights were set, how the strategies were selected, and to what extent these method parameters are specific to a given schema or a given set of test objectives. In contrast, GödelTest can automatically tune the generation process to approach test targets such as comprehensibility or coverage, and the XSD-to-GT connector can use any XML Schema definition as input.

Rather than starting from a formal description of valid XML messages such as XML Schema definitons, Offut and Xu [21] recorded actual messages and then perturbed the data in them based on message grammar rules to ensure syntactic validity. They then used the set of generated XML messages as a test suite for web services. Recently, de Melo and Silveira [22] improved this approach by adding mutation operators and boundary values extracted from XML Schema facets. They also review similar approaches that manually craft specific data and structure mutation operators that are be applied to XML messages. Harikov et al. [23] describe XMLMate, a search-based tool for generating XML inputs. XMLMate uses XML schemas to ensure validity, can utilise existing test data to improve the realism of the generated inputs, and applies evolutionary search—using XML-specific mutation and crossover operators—to optimise coverage of the code.

In contrast to this body of work, GödelTest is a general test generation tool which needs no data-specific operators nor rules in order to construct a large number of syntactically valid test data instances. However, we note that a benefit of mutating recorded or supplied input examples is that the generated test data can be expected to be more realistic since it stays closer to actual, human-created examples. This will likely also increase comprehensibility of the generated data even though it has not been explicitly evaluated.

5.3. Generating Realistic Test Data

Related to the notion of comprehensibility is whether generated test data can be considered realistic or not. McMinn et al. [24] argue that automated test data generation techniques, in which the objective is typically to only optimise coverage, produce test data that is not realistic in the sense that it would be unlikely to be encountered in real-world use of the software. They argue that such inputs add a qualitative 'human oracle cost'—in contrast to quantitive costs such as the number of test cases—since the test cases are more difficult to understand, and propose that applying even limited knowledge as to the SUT's operational profile to the generation process will reduce this cost. An example of such an approach is that of Afshan et al. in which a model of natural language is incorporated into a technique for generating string inputs [25].

Bozkurt and Harman make a similar argument: that syntactically valid test data is not enough; for test data to be realistic it also must be semantically valid [26]. As an example they note that a realistic ISBN number should not only be of the correct format, it should also correspond to a real-world book. They then describe an automated system that can search for web services and use them to learn sets of realistic test data that can later be used for testing web services.

McMinn et al. [27] utilise the internet to locate realistic test data for string types. Web pages are retrieved from search engines using queries constructed from identifiers in the SUT's source code such as parameter, method, or class names. Strings in these web pages are then used to extend the search-based generation of test data for the SUT.

Both these techniques focus on generating realistic inputs for string-like data types such as URLs, ISBNs and ZIP codes that may be sensibly bounded in size. In contrast, the XML data types considered in this paper are more complex tree-like data types which are unbounded in size. For this reason, the approach we describe in this paper considers comprehensibility in terms of the size of XML input, rather that how semantically realistic the input is. As discussed above, optimising this latter aspect would be a useful extension to our approach.

6. Conclusion

We have described an automated process for generating XML test inputs starting from an XML Schema definition. An empirical study using MathML showed that, in a matter of minutes, test sets could be generated that demonstrate code coverage approaching that of a manually-derived test set.

We also argued that XML test inputs must be sufficiently complex to exercise the software-under-test efficiently, but must also be comprehensible for a human, especially when no automated oracle exists for the tests. We proposed a metric for evaluating the comprehensiblity of XML and showed that the GT-NMCS algorithm could use this metric to sample XML test inputs that are more comprehensible than inputs generated by random sampling.

As future work, we intend to apply a similar 'connector' approach in order to automatically construct GödelTest generating programs from the specifications of other data structures, such as JSON and MDE models. There is also scope to investigate other methods of optimising the GödelTest generation process: GT-NMCS, while generating more comprehensible test data than random sampling, nevertheless generated test inputs for which the metric was relatively distant from the target. One possible solution is to use metaheuristic optimisation of the choice model prior to sampling GT-NMCS. A future enhancement is to extend to the technique to improve the comprehensibility of not only the structure of generated XML inputs in terms of size, but also the realism of the data itself. Finally, we would like to validate empirically whether our proposed comprehensibility metric reduces the cost of a human oracle in practice.

Acknowledgments

This work was funded by The Knowledge Foundation (KKS) through the project 20130085, Testing of Critical System Characteristics (TOCSYC).

References

[1] R. Feldt, S. Poulding, Finding test data with specific properties via metaheuristic search, in: Proc. of 24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013), IEEE, 2013, pp. 350–359.

[2] S. Poulding, R. Feldt, Generating structured test data with specific properties using Nested Monte-Carlo Search, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014), 2014, pp. 1279–1286.

[3] R. Debreceny, G. L. Gray, The production and use of semantically rich accounting reports on the internet: XML and XBRL, International Journal of Accounting Information Systems 2 (1) (2001) 47–74.

[4] M. Masmano, I. Ripoll, A. Crespo, J.-J. Metge, Xtratum: a hypervisor for safety critical embedded systems, in: 11th Real-Time Linux Workshop, 2009.

[5] G. Fraser, A. Arcuri, Handling test length bloat, Software Testing, Verification and Reliability 23 (7) (2013) 553–582.

[6] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A fast dynamic language for technical computing, arXiv preprint arXiv:1209.5145.

[7] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Computational Intelligence and AI in Games 4 (1) (2012) 1–43. doi:10.1109/TCIAIG.2012.2186810.

[8] D. Fallside, P. Walmsley, XML Schema Part 0: Primer Second Edition, Tech. rep., W3C (2004).

[9] D. Lin, LightXML, `https://github.com/lindahua/LightXML.jl` (2014).

[10] M. Berger, JEuclid: Java-based mathml rendering, `http://jeuclid.sourceforge.net/`, version 3.1.9 (February 2010).

[11] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, S. Watt, Mathematical Markup Language (MathML) Version 2.0 (Second Edition), Tech. rep., W3C (2003).

[12] P. Duchon, P. Flajolet, G. Louchard, G. Schaeffer, Boltzmann samplers for the random generation of combinatorial structures, Combinatorics, Probability and Computing 13 (4-5) (2004) 577–625.

[13] M. R. Hoffmann, EclEmma: Java code coverage tool for eclipse, `http://www.eclemma.org/`, version 2.3.1 (May 2014).

[14] M. Beyene, J. Andrews, Generating string test data for code coverage, in: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST 2012), 2012, pp. 270–279.

[15] S. Poulding, R. Alexander, J. A. Clark, M. J. Hadley, The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2013), 2013, pp. 1477–1484.

[16] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, D. Marinov, Test generation through programming in UDITA, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, 2010, pp. 225–234.

[17] K. Claessen, J. Hughes, Quickcheck: A lightweight tool for random testing of haskell programs, in: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 2000, pp. 268–279.

[18] A. Bertolino, J. Gao, E. Marchetti, A. Polini, Automatic test data generation for XML schema-based partition testing, in: Proceedings of the second international workshop on automation of software test, IEEE Computer Society, 2007, p. 4.

[19] C. Bartolini, A. Bertolino, E. Marchetti, A. Polini, WS-TAXI: A WSDL-based testing tool for web services, in: Software Testing Verification and Validation, 2009. ICST'09. International Conference on, IEEE, 2009, pp. 326–335.

[20] A. Bertolino, F. Lonetti, E. Marchetti, Systematic XACML request generation for testing purposes, in: Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on, IEEE, 2010, pp. 3–11.

[21] J. Offutt, W. Xu, Generating test cases for web services using data perturbation, ACM SIGSOFT Software Engineering Notes 29 (5) (2004) 1–10.

[22] A. C. de Melo, P. Silveira, Improving data perturbation testing techniques for web services, Information Sciences 181 (3) (2011) 600–619.

[23] N. Havrikov, M. Höschele, J. P. Galeotti, A. Zeller, XMLMate: evolutionary XML test generation, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 719–722.

[24] P. McMinn, M. Stevenson, M. Harman, Reducing qualitative human oracle costs associated with automatically generated test data, in: Proceedings of the First International Workshop on Software Test Output Validation, ACM, 2010, pp. 1–4.

[25] S. Afshan, P. McMinn, M. Stevenson, Evolving readable string test inputs using a natural language model to reduce human oracle cost, in: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, IEEE, 2013, pp. 352–361.

[26] M. Bozkurt, M. Harman, Automatically generating realistic test input from web services, in: Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on, IEEE, 2011, pp. 13–24.

[27] P. McMinn, M. Shahbaz, M. Stevenson, Search-based test input generation for string data types using the results of web queries, in: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, 2012, pp. 141–150.